Ammar Abu Shamleh (21521274)
Pradyumn Vij (21469477)

Parallelisation of column-wise matrix collisions

HIGH PERFORMANCE COMPUTING – PROJECT 1

1.0 Problem

A large set of floating point numbers reside in data that is ordered in rows and columns, described as lines in text files.

Each column must have blocks of 4 numbers extracted. A quartet of numbers are termed a block if they are all within the same arbitrary range of each other. The arbitrary range will be referred to as the "DIA". For example, in the trivial case where a column consists of {0.0, 0.1, 0.25, 0.5, 0.6}, the blocks present when the DIA of 0.5 are {0.0, 0.1, 0.25, 0.5} and {0.1, 0.25, 0.5, 0.6}.

Each row has been provided with a unique large integer, and based on project statements, a Block's signature is generated by adding each the four element's row keys together. An assumption that this signature is unique for each block (within the column in which it's found) is based on the project documentation provided.

The task is to find all collisions in the data set. A collision is defined by blocks with the same signature in different columns.

1.1 Problem Statement

Implement the task described in section 1.0, using the OpenMP library to optimise task execution.

1.2 Problem Approach

The problem was approached by first implementing the desired utility described in the most obvious manner (i.e. a brute-force approach). This first approach was performed sequentially, and following this, various avenues for parallelization on the brute-force sequential code were explored. Brute-force code typically doesn't have tremendous dependencies across loop iterations, so the initial parallelization was predicted to be a simple task. This would later be followed up by algorithmic redesign on the original sequential code, and then parallelization of the new algorithmically optimized code. This

allowed for an interesting comparison of the potential speed-ups gained by parallelization vs. Algorithmic redesign.

Optimising for OpenMp required careful design by the programmers, so as to eliminate dependencies across tasks. The more dependencies there are between different tasks being performed, the harder it is to parallelize the execution of those tasks. Additionally, excessive usage of thread synchronization mechanisms, such as barriers, can not only reduce speed-up factors, but even result in slower code than the original sequential version.

The problem was split into 3 main tasks:

1. File input (i.e. reading all matrix data and key data into data structures within the program)
2. Block generation. This involved proceeding through data and finding all blocks
3. Collision detection among located blocks

2.0 File input ,data parsing and data structures

The keys were supplied in a file named "keys.txt". The floating point data (i.e the data matrix) was provided in a file named "data.txt". The first task faced in approaching this project was to read all data residing in these two files into internal data structures for use in the program. While this could easily be accomplished using static arrays, the task was instead approached using dynamically allocated arrays. The advantage of this approach is that it doesn't tie the code to the specific data set that has been provided, and makes the code more robust for dealing with other data set sizes.

2.1 Data Input

The floating point numbers provided in data.txt were not prescribed to any specific decimal place. While the example data set provided contains numbers to 6 decimal places, no inherent restrictions were provided in the project brief. To ensure robustness, the data type of double was used for extra precision, in the case that more precise results are required from future data sets.

The keys were chosen to be of type long long. The reasoning here was to avoid the large integers causing arithmetic overflowing on any 32 bit (or lower) architecture. The numbers themselves, within the provided data set, are fine as integers. But when added to create a signature, they numbers are at risk of causing overflow errors.

In the following sections the variable “n” refers to the number of inputs values per text file.

2.1.1 readMatrix - O(n)

The function “readMatrix” counts the number of rows and columns of the matrix in “data.txt” while dynamically allocating memory and assigning values to internal data structures. A new row is defined by a new line in the text file and a new column for each comma separated number in a line.

2.1.2 readKeys - O(n)

The function readKeys reads the "keys.txt" file into an array of long long of size ROWS, defined in function "readMatrix" (see section 2.2). The function readKeys runs in time and space complexity O(n).

2.2 Structs

2.2.1 Block

The Block struct holds the signature of a unique block of 4 elements from any one column in type long long. The entire program hinges on the assumption that signatures are unique for every unique quartet of elements found (within any given column; signatures can occur repeatedly across columns).

The column from which the block originates is also stored.

In addition, in order to aid with the post-processing merging of blocks, each block's unique combination of 4 row-numbers are stored as an array of integers.

2.2.2 Collision

The Collision struct is required to accumulate all possible collisions prior to post processing.

An array of integers named "columns" holds the columns in which the collision's blocks were found. For example, if 3 blocks with the same signature are found across columns 4, 6 and 15, then the array will hold these 3 column numbers. The integer value "numBlocksInCollision" holds the number of values in the array (also equal to the number of blocks that were found as part of the collision).

2.2.3 Pair

The Pair struct is required by the optimised sequential block generation code to hold the row key value, in the form long long, and an integer value describing the original row index. The requirement of this struct is to remember these values after sorting, which is required by the approach developed for the optimised block generation algorithm. This struct is also used in the parallelised block generation code.

2.3 Memory management

In the process of optimising the multithreaded performance using the OMP library, several hurdles were encountered in the form of memory/communicational bottlenecks to performance. Firstly, it was discovered that the array of block pointers (the database of all located blocks, which is populated by the block generation functions) was causing a high number of cache misses and very few hits. These memory issues were only exacerbated in the multi-threaded code.

The reason for this is that an array of block pointers means that each subsequent access to the next block will be likely to incur a cache miss, as the blocks are not contiguous in memory. While the pointers themselves are contiguous in memory, each pointer points to a dynamically allocated space in memory, and these spaces are all very likely to be

disparate. As such, blocks themselves are not contiguous in memory, and a loop through them would incur lots of cache misses. Cache unfriendly code such as this is only worsened by attempts to multi-thread, as effective parallelization requires careful management of memory and cache.

The result was a very slow program, or one that did not run at all, based on the operating system/architecture used.

This issue was targeted by instead creating an array of block structs, rather than an array of pointers to block structs. Thus, subsequent blocks became contiguous in memory, and the cache unfriendliness was resolved.

Another memory issue identified was the storage method used for the data matrix. The data was stored as it was represented in the files, meaning that the rows of the data file were stored as rows in the 2d array of doubles that was used within the program to represent the data. Given that the work in each column is independent of work done in other columns, the outer loop of the block generation code was on the columns of this 2d array of doubles. This introduced severe memory issues, as a loop over array columns typically incurs a huge number of cache misses.

Consider:

```
for(int col=0; col<MAX_COLS; col++) {
  for(int row=0; row<MAX_ROWS; row++) {
    data[row][col]
  }
}
```

Code such as the above will run very poorly. The reason is that subsequent elements of array rows are stored contiguously in memory. This means that, given large enough MAX_ROWS and MAX_COLS sizes, the above code will almost certainly cause a cache miss at every subsequent memory access.

An identical issue was encountered in this project, in that the outermost loop was over columns of the data matrix, rather than rows (as the blocks are all found within a column). To rectify the issue, the data matrix was transposed, so as to represent the data's columns as rows inside the double array. Applying this to the above code snippet, the resulting code would look like:

```
transpose(data);
for(int col=0; col<MAX_COLS; col++) {
  for(int row=0; row<MAX_ROWS; row++) {
    data[col][row]
  }
}
```

Thus the logical operation of the internal contents of the loop is maintained, and at the same time, the code is substantially more cache-friendly.

After transposing the data matrix, both the sequential and parallel code for block generation were found to run several times faster, as a result of the improved cache friendliness of the code.

2.4 Managing access to shared data during parallelisation

One issue that needed to be considered was the access to shared data structures in multi-threaded code. Having multiple threads carelessly access shared data-structures, and in particular, attempt to write to shared data-structures, simultaneously, can introduce several issues. In general, managing access by multiple threads to shared data was an issue that required careful examination and thought.

There were two issues for consideration here:

1. Ensuring that thread's simultaneous access to shared read-only data doesn't introduce too many cache misses
2. Ensuring that threads can simultaneously write blocks and collisions they find without memory clashes

2.4.1 Managing simultaneous access to shared read-only data

As threads work on shared data, it is important that the data is split carefully among the working threads, so as to maximize cache-hits. As an example, if an array has 4000 rows (as in this case, with the data set), and the data is split carelessly into blocks 0-1000, 1000-2000, 2000-3000 and 3000-4000, then such parallelization will incur a large number of cache misses.

A more effective method of splitting the data was required, so as to maximise temporal and spatial locality within the code. A better approach found was to have the data split using thread numbers and number of threads instead. This will be examined further, at a later stage of the report.

2.4.2 Allowing multiple threads to write to a block database simultaneously

As blocks are found, they are to be written to a block database (i.e. an array of block structs). When multi-threading, there will be multiple threads running simultaneously, and having multiple threads write to a shared data space in parallel could introduce severe issues. As such, an alternative approach was found:

Each thread will maintain its own private database of blocks, and will add any blocks it finds to this private database. When a thread has finished its processing work, the private database it created will be merged with a collective, shared database, which is the final storage location for any blocks found. This merging step takes place inside a critical region of code, such that only one thread can have write access to the shared

database at any one time. While the critical region does introduce overheads that may damage performance, the parallelization still managed to attain excellent speedups. An identical approach was used for the storage of collisions.

3.0 Identifying Blocks in data

A function (findBlocks) was written for the 2d array of floating point number data (dataMatrix) to iterate over each column and find all Blocks (within each column) with any given DIA. When a block is found it is stored in an array of Blocks.

3.1 Sequential brute force - O(kn⁴)

The starting approach to the project was a simple brute-force, naïve algorithm to find blocks. The algorithm simply contains a loop over all columns, with each iteration making 4 nested loops over all rows to find all blocks. The array of blocks is dynamically reallocated to fit each new block that is found (preventing memory wastage by statically allocating too large an array, or a segmentation faults by allocating too small an array). The approach was akin to:

```
//A loop over all columns
for(int col=0; col<MAX_COLS; col++) {
  for(int r1=0; r1,MAX_ROWS; r1++) {
    for(int r2=r1+1; r2<MAX_ROWS; r2++) {
      for(int r3=r2+1; r3<MAX_ROWS; r3++) {
        for(int r4=r3+1; r4<MAX_ROWS; r4++) {
          //Check if all 4 elements are within DIA
          //If so, generate and store a new block
        }
      }
    }
  }
}
```

As can be seen above, the work done in each column is independent. Additionally, the outermost loop on rows (the loop on **r1**) also has iterations that are all independent of one another. Each loop of that outermost loop on rows will find an independent, unique set of blocks. So both the loop on columns, and the loop on **r1**, were identified as potential targets for parallelization. Another note is that because of the way the loop works, with each subsequent index beginning at the value above the previous index (i.e. **r2** starting at **r1+1**), it is guaranteed that no duplicate blocks will be found.

The time complexity of this approach is O(n^4) per column, where n is the number of rows. Thus the overall time complexity is O(kn^4), where k is the number of columns, and n is the number of rows. One limitation with this approach is the fact that it only works for block sizes of 4. A change in the block size would entail changing the code to adapt to the new block size (not a desirable feature for robust programs, where such attributes should be freely modifiable).

3.1.1 Results

Using a processor timer for timing execution was deemed inappropriate, as it does not give a fair description of the task time taken, especially when timing multi-threaded code. An elapsed time approach was required, using the omp.h library's omp_get_wtime function.

With all **500 columns** and a **DIA of 10^{-6}**, this algorithm takes **2 minutes and 43 seconds** to run. When running only the first **499 columns (same DIA)**, the algorithm takes **26 seconds** to complete on the test machine. All **500 columns** were found to have **29,321,366 blocks**, while the first **499 columns** only have **14,863 blocks**. It can be seen here that the last column (the 500th column) contains the vast, vast majority of blocks in the data.

3.2 Multithreaded brute force – $O(kn^4)$, Parallelise by Column

The initial approach to parallelization of the block generation function was to split the work-load by column, i.e. have each thread work independently on a different column. This was considered a viable approach, as the work done in each column is independent of the work done in all other columns. Accomplishing multi-threading in this manner was simple, as the outermost loop iterates over columns (as seen in the earlier image). Thus it was a simple case of parallelizing that outermost loop using omp's provided pragmas. While parallelization was achieved, there were bottlenecks that prevented the approach from achieving a good speed-up factor.

Primarily, a single column was discovered to hold the majority of blocks in the test data set provided.

3.2.1 Limitations

As can be seen in **3.1.1 Results**, column 500 holds a substantial number of blocks. As such, splitting the work load by column was clearly limited by a single thread having to work on a column that is heavily populated in the provided data set. This load-balancing issue resulted in a poor speed-up factor. We felt this accurately portrayed the types of issues that are encountered in parallelizing the processing of real-world data: perfect load-balancing is difficult/impossible to achieve, in many cases.

If the blocks were evenly distributed across columns, this approach would have worked far better.

3.2.2 Results

Unfortunately, due to most of the processing work taking place in the last column, which is still only processed by a single thread, no measurable speed-up was attained by this approach.

3.3 Multithreaded brute force – $O(kn^4)$, Parallelise within Columns

Given that load-balancing issues were faced with the previous approach, a new, revised approach was considered. Instead of splitting work by columns, the work would

instead be split within columns. This would allow for the effective parallelization of data with very uneven distribution of blocks among columns.

So, to address the load imbalance discovered in the previous attempt, parallelization would occur within each column, rather than across the columns. So rather than assigning columns to threads, the outermost loop of the work done **within** each column was distributed evenly among the threads. Thus the parallel region begins and ends within each column.

The column's elements were divided by the number of threads, and each thread would individually find a subset of blocks within a column (updating a thread-private database). These partial sets of blocks would then be merged, within a critical region (so as to avoid race conditions and simultaneous memory accesses by multiple threads to the same region in memory), with a collective, shared database of blocks. Thus the workload within each column would be split among the multiple active threads.

The work allocation was done using **omp_get_thread_num()** and **omp_get_num_threads()**. Each thread would begin work on an index (over the outermost loop over rows) equal to its thread ID number, and this index would be incremented by the number of active threads. Thus if 4 threads are active, thread 0 begins work at index 0, and increments its index by 4 on each subsequent iteration. This approach is much more cache-friendly then splitting the loop into contiguous chunks for each thread to work on (i.e. assigning thread 0 iterations 0-1000, thread 1 iterations 1000-2000, thread 2 iterations 2000-3000 etc.). The reason is that with this approach, the threads all remain within roughly the same region of the data as they process it, and thus referential locality of the code is maximized, and so too is the cache-hit to cache-miss ratio.

The method looked something like:

```
//A loop over all columns
for(int col=0; col<MAX_COLS; col++) {
  #pragma omp parallel
  {
  //BEGIN PARALLEL REGION
          Block *privateDatabase = malloc(......);
          int ID = omp_get_thread_num();
          int nthreads = omp_get_num_threads();
          for(int r1=ID; r1,MAX_ROWS; r1 += nthreads) {
            for(int r2=r1+1; r2<MAX_ROWS; r2++) {
              for(int r3=r2+1; r3<MAX_ROWS; r3++) {
                for(int r4=r3+1; r4<MAX_ROWS; r4++) {
                  //Check if all 4 elements are within DIA
                  //If so, generate and store a new block
                }
              }
            }
          }
  //END PARALLEL REGION
  }
}
```

This approach to parallelisation was investigated to try to introduce multi-threading on the heavily populated 500th column, which was found in prior approaches.

One downside to the approach is that with parallel regions beginning and ending per column, the overhead of creating new threads, and the overhead of encountering the aforementioned critical region, are now encountered multiple times (once per column for generation of new threads, and once per thread per column for critical region). Thus, while more even load-balancing is attained, it comes at the cost of increased overheads. For columns with very, very few blocks, this could actually result in worst performance with the parallel code (as there is not enough computation work being done to justify the overheads involved in the parallelization).

3.3.1.1 Special Case Observations

While the basic approach described above worked extremely well for the first 499 rows, it didn't attain any speed-up (and in fact caused slowdown) for the last column.

This was addressed by writing a special case for the last column, where by the last column would be split among a set of threads using tasks. A single thread would generate tasks, each of which perform processing on a certain chunk of the last column. These tasks are added to a task pool that is worked on by any available threads. This improved performance for the last column compared to the simple split of the for-loop work, but there was still slight slowdown compared to the sequential code.

On further examination, it appeared that the load balancing problems that were occurring at a column level were also occurring at a section level, as there were far more blocks in the first and second sections of the column than the rest, leaving idling threads. Thus the size of chunks generated by tasks had to be revised, to more evenly distribute work among threads and maximize core utilization.

This problem was targeted by reducing the chunk size of each task, and thus providing more threads for the first 2000 elements. While this revision improved load balancing, it actually reduced performance for some reason. It seems that when multiple threads were running simultaneously on the last column, they would individually each work slower than when fewer threads were running.

These observations strongly suggested memory access problems; that each thread's execution was interfering (at a cache level) with the other active threads. After multiple failed attempts to rectify the issue, the last column was simply discarded, and the program would hence only run on the first 499 columns.

3.3.1.2 Special case limitations

The most obvious limitation is that this approach (with the tasking on the last column) would only work on this particular data set.

3.3.1.3 Special case results

Memory management issues were identified. Load-balancing is a difficult issue to completely resolve for unpredictable, real-world data

3.3.2 Introducing transposition

To address the memory mismanagement issue described earlier (with the outermost loop being over columns), the data matrix was transposed so that the columns are stored in rows, thus reducing cache-miss frequency substantially.

Matrix transposition only costs O(n) in time complexity.

3.3.2 Limitations

One limitation is the issue described earlier, concerning the parallel region beginning and ending once per column. This incurs lots of overheads.

A second issue is that each thread requires its own private database of blocks (to avoid concurrency issues by writing to a shared database). A merge of these databases would be required at the end of each thread's work in a column. This merging would take place in an omp critical region, which is damaging to performance. The critical region would be encountered k * t times, where k is the number of columns and t is the number of threads, an example would be 4 threads with 500 columns resulting in 2000 encounters.

3.3.3 Results

The result on the first **499 columns** (excluding the populous column), **before** matrix transposition is a multithreaded run time of block-generation that takes an average of **12.406089 seconds** versus an average of approximately **26 seconds** for the base sequential code.

This is a speed-up of a factor of **2**.

After transposing the data matrix, the sequential run-time dropped down to about **19.172642 seconds** (from **26 seconds**), and the parallel run-time dropped down to about **4.854188 seconds**, achieving a speed-up factor of **4** with the parallel code. Thus the cache-miss resolution increased the parallel speed-up factor significantly (roughly doubled it), reinforcing the idea that memory management is crucial for effective parallelization.

3.4 Sequential Sorted Combinatorics – $O(kn^4)$

An algorithmic improvement of the brute force would clearly improve the run time of finding the blocks. This particular method uses the dataMatrix's columns (now rows after transposition).

For each column, a temporary array of Pairs is created. Where a Pair consists of a value, which is the actual value in the data Matrix (a double), and a key from the key array (a long long). This temporary array of pairs is sorted according the value (double) of the pairs.

The next step is to have a starting pointer and an end pointer defining a region of size 4 or more where the difference between the upper bound minus the value of the lower bound is within DIA. All the values between the upper and lower bounds can construct valid unique combinations of size 4 to create blocks. Thus only unique blocks are ever calculated.

The time complexity of sorting is O(nlgn).

The cost of iterating across the column is O(n).

The cost of finding all unique combinations in a window is $O(w^3)$, where w is the number of elements in the window (greater than 4).

The resultant complexity is $O(k(nlgn + nw^3))$.

In the very worst case, every element in the array is within DIA which makes the time complexity $O(kn^4)$. However the average case is O(kn + B) where B is the number of valid blocks in the data provided.

3.4.1 Results

All following results are with a **DIA** of **$2.5 * 10^{-6}$**

The algorithm (sequentially) finds all blocks in the first **499 columns** in **0.354579 seconds**

When trialled over **all columns** (inclusive of the populous **500th column**), an average run time of **0.906457** seconds was achieved during testing.

These are certainly impressive results, and indicate that the worst case is highly unlikely.

Compared to the brute force sequential algorithm, the optimized code (on this data set) for the first **499 columns** ran an average of **54.07 times faster**. In addition, it runs **13.689 times faster** than the best multi-threaded attempt (which took **4.85 seconds**) on the brute-force code.

There were several interesting observations that were derived from this improvement, concerning the efficiency improvements that can be gained by parallelization vs. Algorithmic improvement. Parallelization focusses on using hardware to split existing workload among multiple cores; the instruction count (IC) of the program remains unchanged. Thus runtime is reduced by using multiple cores for execution. Conversely, algorithmic improvements focus on minimizing the instruction count (IC) of the program through logical redesign. While parallelization speed-up factors are severely limited by the number of cores available, and the overheads involved in creating new threads, radical algorithmic redesign can (as shown in this example) drastically improve performance of a program. This introduces several interesting questions about whether it is better to focus on designing code to be parallelizable, or whether it is instead more efficient to focus on major algorithmic improvements, at the cost of parallelizability. Ultimately, we felt that the answer to this question is severely context dependent, and will change depending on which approach gets better performance for the specific problem being solved.

3.5 Multithreaded Sorted Combinatorics – $O(kn^4)$

Given that the optimized code still treats columns independently, the best approach to parallelization was actually the simplest: simply parallelizing the outer for loop that loops over the columns.

Simply creating an **omp parallel for** pragma around the outermost (loop over column) for-loop was able to yield an impressive speed up, as each column was being dealt with

completely independently, while the computations within the inner for loops are too dependent to easily multithread. The complexity stays the same as above but the multithreading directly addresses the k portion of the complexity (i.e. the work-load of k columns is split among multiple cores, reducing run-time).

3.5.1 Results

Results below are with a **DIA of 2.5 * 10^{-6}**

For the first **499 columns** the algorithm takes an average of **0.097343 seconds** to execute. Compared to the sequential sorted combinatorics algorithm, this provides a speed-up factor of **3.64257**.

For all **500 columns** the algorithm takes an average of **1.707430 seconds** to execute. This actually introduces slowdown compared to the sequential code. It runs roughly **1.9 times** slower than the sequential optimised code for all **500 columns**.

The inclusion of the load-unbalancing 500th column is a clear example of where a multithreaded approach can be detrimental based on the data set. For situations when the load is heavily favoured to one column (such as this data set), there is a reduction in speed by a factor of 2. However, in cases where the balance is spread over multiple columns, the increase in speed can be significant, as shown above, where it's sped up close to **3.5 times.**

4.0 Identifying Collisions amongst Blocks

After finding and storing all Blocks in a block database, the blocks involved in collisions must be identified, within the function findCollisions. This function's purpose is to determine which signatures exist in more than one column, and then determine the set of columns over which each signature involved in a collision occurs.

4.1 Sequential Brute Force – O(n^2)

This naïve method involves creating a Boolean array of the same numerical size as the array of Blocks. The function is to check each Block's signature and check it against all other blocks in the array of blocks. If other blocks with matching signatures are found, the number of occurrences (i.e. the number of blocks with the same signature) is recorded and stored in a Collision, along with the set of column numbers in which the same signature was found. This collision is then stored in an array of Collisions. In the process, any blocks found in collisions are marked as true in the relative position within the Boolean array, to avoid repetitions. i.e if blocks 1 ,5 and 7 are found in a collision, then positions 5 and 7 in the Boolean array are marked as true, so as to avoid blocks 5 and 7 later being found in a duplicate collision.

The time complexity of this function is O(n^2), where n is the number of blocks. The space complexity required is O(B + b + c), where B is the number of Blocks; b is the number of Booleans (equal to the number of Blocks, but smaller as storing a Boolean rather than a Block); c is the number of collisions.

4.1.1 Results

Results below are with a **DIA of 2.5 * 10^{-6}**

For the first **499 columns**, to find all collisions the algorithm takes an average of **71.75 seconds** to finish.

For all **500 columns** the algorithm failed to finish.

4.2 Multithreaded Brute Force – O(n^2)

The method described in section 3.3 is similar to the approach taken to optimise the performance of brute force in this particular situation.

For the first 499 columns, the algorithm takes an average of **20.374063 seconds** to finish. A speed-up factor of **3.52**

With all **500 columns**, the algorithm failed to finish.

4.3 Sequential Sorted Blocks – O(nlgn)

Similar to block generation, the next step was an algorithmic redesign on the base sequential code. This time, the algorithmic approach to the collision detection within the database of blocks was substantially improved

The most optimal solution found is to sort the array of Blocks by signature, which costs time and space O(nlgn).

The next step is to iterate through the sorted blocks looking for identical signatures. Now that the array is sorted by signatures, all identical signatures (and thus all collisions) will be grouped up, and easy to find. If more than 1 contiguous identical signature in the array is found, all block's columns in the contiguous signature instance are added to a single collision. This second part of the solution also builds the array of Collisions and costs O(n).

4.3.1 Results

Results below are with a **DIA of 2.5 * 10^{-6}**

For the first **499 columns**, to find all collisions the algorithm takes an average of **0.060873 seconds** to execute, providing a speed-up factor of **1179** compared to the sequential brute-force code. Similarly, compared to the parallel brute-force code, the speed-up factor here is **334.6**.

For all **500 columns**, to find all collisions the algorithm takes an average of **9.000617 seconds** to execute.

This again illustrates how algorithmic redesign of code can achieve substantially higher speed-ups than parallelization alone can ever provide.

4.4 Multithreaded Sorted Blocks – O(nlgn)

Parallelizing the optimised collision detection algorithm proved a more involved task than all earlier attempts at parallelization. The reason is that there were more complex dependencies within the for-loop. In earlier stages, the outermost loop's iterations were always independent of one another, allowing for simple parallelization.

However, the issue faced with this new algorithm was that there was only one for-loop over the array of blocks (now sorted by signatures, thus collecting collisions together within the array). The problem here is simple parallelization of the for-loop would destroy the correctness of the program, as the array was effectively split into variable sized chunks, with each chunk (a chunk being a segment of the array where the signatures of the blocks are all equal) represents a collision. Thus each chunk can only be worked on by one thread at a time; given that the chunks are variable sized, and could begin and end anywhere in the array, this made parallelization a complex task.

Consider this example:

```
Block Array
Block 1: Signature 11201
Block 2: Signature 11201
Block 3: Signature 11201
Block 4: Signature 11201
Block 5: Signature 11402
Block 6: Signature 11504
Block 7: Signature 11504
Block 8: Signature 11705
Block 9: Signature 19120
......
......
......
```

This effectively represents the manner in which the algorithm proceeded. The issue here is that the loop over these blocks can't be parallelized by a simple **omp for pragma**, as a collision might be cut in half, with one half allocated to one thread, and the other half allocated to another thread. This would then result in an incorrect result. For instance, in the above example, careless parallelization of the for-loop could split the first collision (on signature 11201) among 2 threads, thus hampering correctness of the program.

The solution found was to use tasking to split the work up. A single thread splits the array into multiple roughly evenly sized chunks, and ensures that each chunk begins and ends on a 'collision boundary' i.e. no collision is chopped up between two chunks. The single thread then generates tasks for the collision detection within each of these chunks, and these tasks can then be worked on independently and simultaneously by multiple threads, which will each develop their own partial database of collisions over the chunk that they have been assigned.

These partial databases are then merged into a single complete database inside a critical region (very similar to how the partial block databases are merged).

4.4.1 Results

Results computed with a **DIA of $2.5 * 10^{-6}$**

Even with the added overhead of the critical regions, a significant speed-up factor was still achieved.

For the first **499 columns** the algorithm takes an average of **0.031493 seconds** to execute. This achieves a speed-up factor of **1.93**.

For all **500 columns** the algorithm takes an average of **3.411191 seconds** to execute, providing a speed-up of **2.64.**

One observation made here is that the speed-up factor attained by parallelizing the optimised collision detection is not as high as the speed-up factor attained by parallelizing the brute-force collision detection. The speed-up factor attained (on **499 columns**) through parallelization of the algorithmically optimized code was roughly **1.93**, in this example. Conversely, the speed-up factor attained through parallelization on the brute-force collision detection code was roughly **3.52**. This is a result of the algorithmically optimized code being less parallelizable, due to the complex dependencies that are introduced into the for-loop by the new approach. However, given that the algorithmic optimization provides such a substantial speed-up factor (through a radical reduction to the program's instruction count, by reducing complexity from $O(n^2)$ to O(nlogn)), we believe that sacrificing a bit of parallelizability for the incredible improvement to performance is a very worthy trade-off.

A potential reason why the speed-up factor here is not as high as with the brute-force collision detection could be a result of the inherent cache-unfriendliness of the way the data is being split. The block database is split into roughly equal-sized chunks, which are allocated to threads. The issue is that (due to the nature and complexity of the sequential algorithm), these chunks are split in a manner that increases the frequency of cache-misses during execution. i.e. one task may operate on a chunk from 0-2002, another task from 2002-3007, another from 3007-4009 etc. This manner of splitting work was previously described (within this report) as being cache unfriendly, due to not exploiting spatial locality. Despite this issue, there is no other effective way to split the collision detection up among multiple threads. Consequently, the multi-threaded performance suffers, and the speed-up factor is diminished.

This example (along with the similar algorithmic optimization of the block generation) demonstrates the inherent bottleneck in parallelization compared to algorithmic redesign: parallelization speed-ups are severely limited by the physical number of cores in the machine, which, on average modern machines, is only 4. In addition, parallelization is also critically dependent on careful memory management, and performance gains can be damaged by too much synchronization or too many overheads in parallel code. Thus, if radical algorithmic improvements can be made to code, this is often a better starting point for performance improvements than parallelization alone.

5.0 Block Merging

Quite late in the project it was found that a post processing task was required to merge blocks using the original row indices. The task required some slight changes to existing structures to accommodate the new data that would be required for the step. While some working, reasonable code was developed to perform this process, it was found to simply run too slowly (O(n^2) on all collisions), and actually proved to be a significant

bottleneck to the program's overall performance, as it occupied the majority of the run-time. As such, the code, while present, is not used, and remains commented out. Given time and resource constraints, a more intelligent, more optimised algorithm than the brute-force O(n^2) algorithm was not pursued.

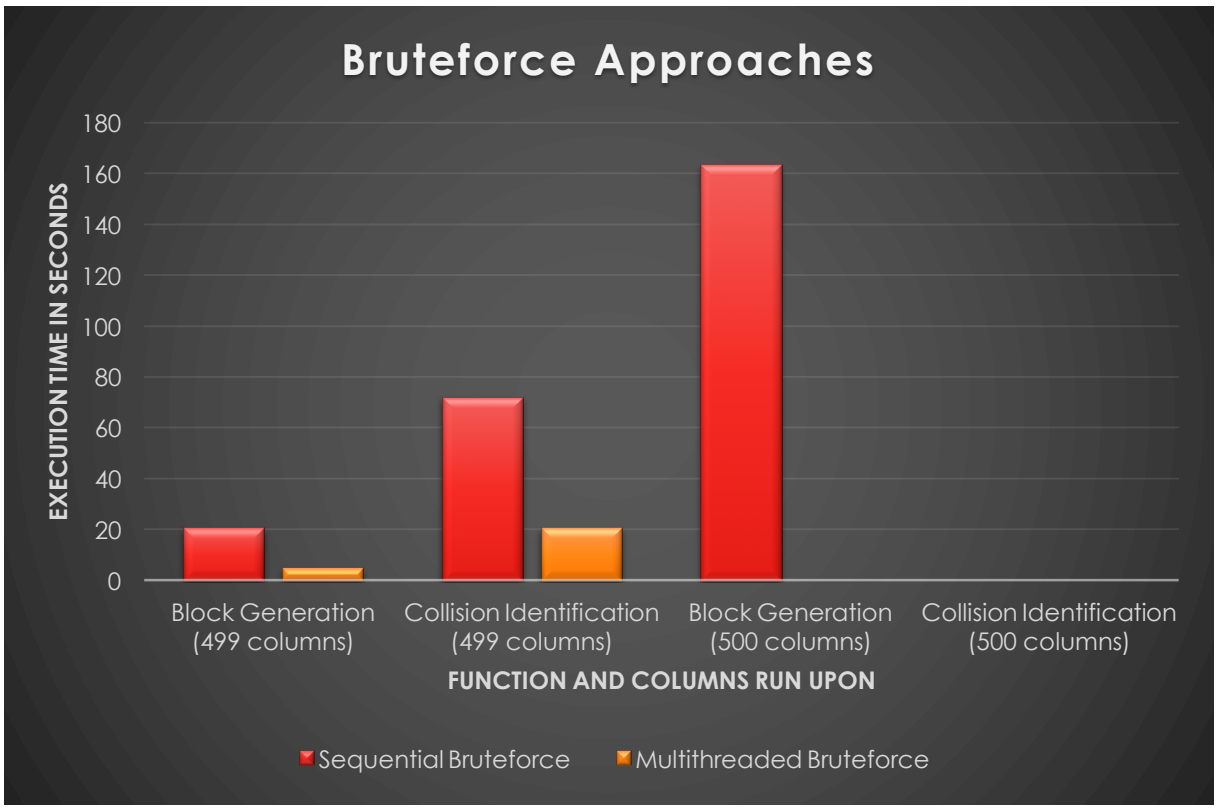
6.0 Experimental Data Analysis and Conclusions

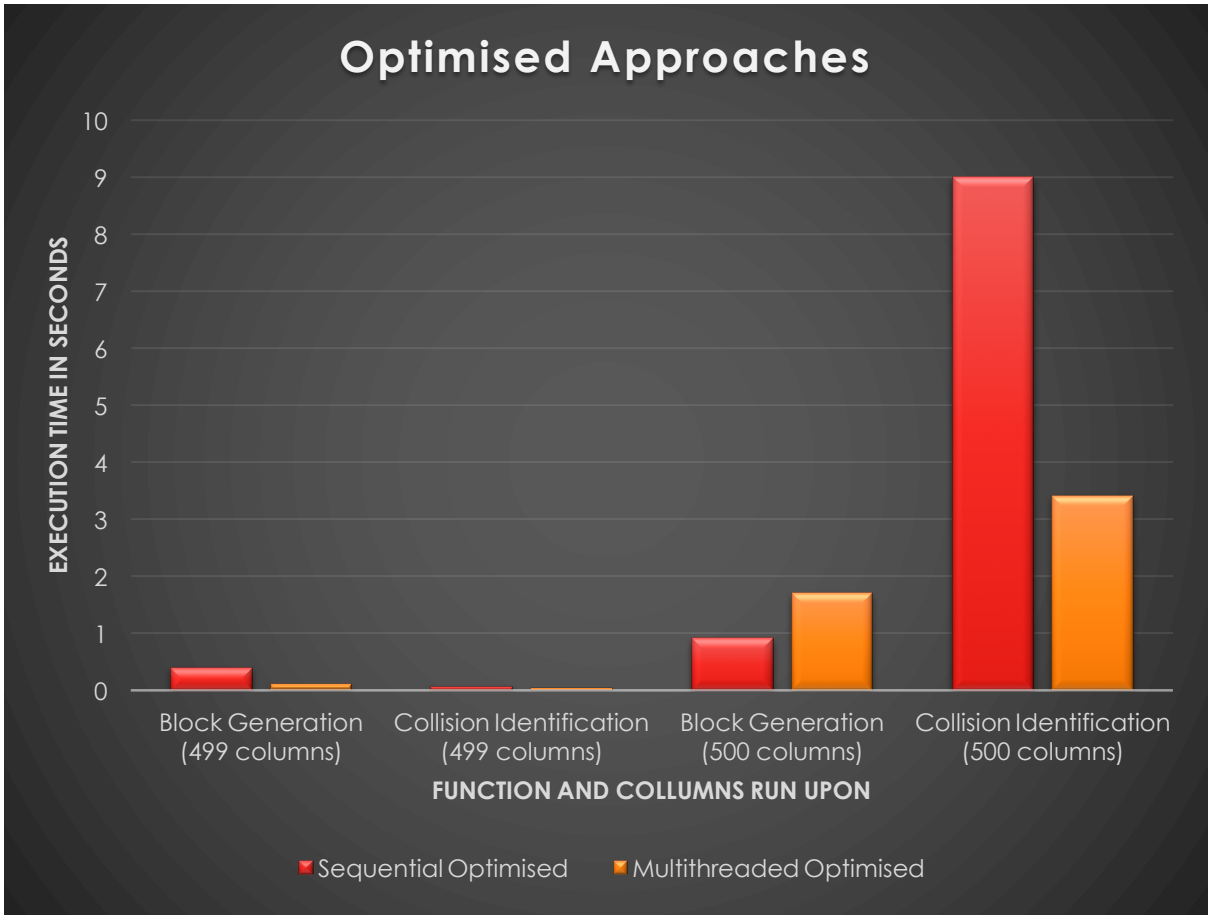
Table 1: Execution Times (best times achieved are in bold)

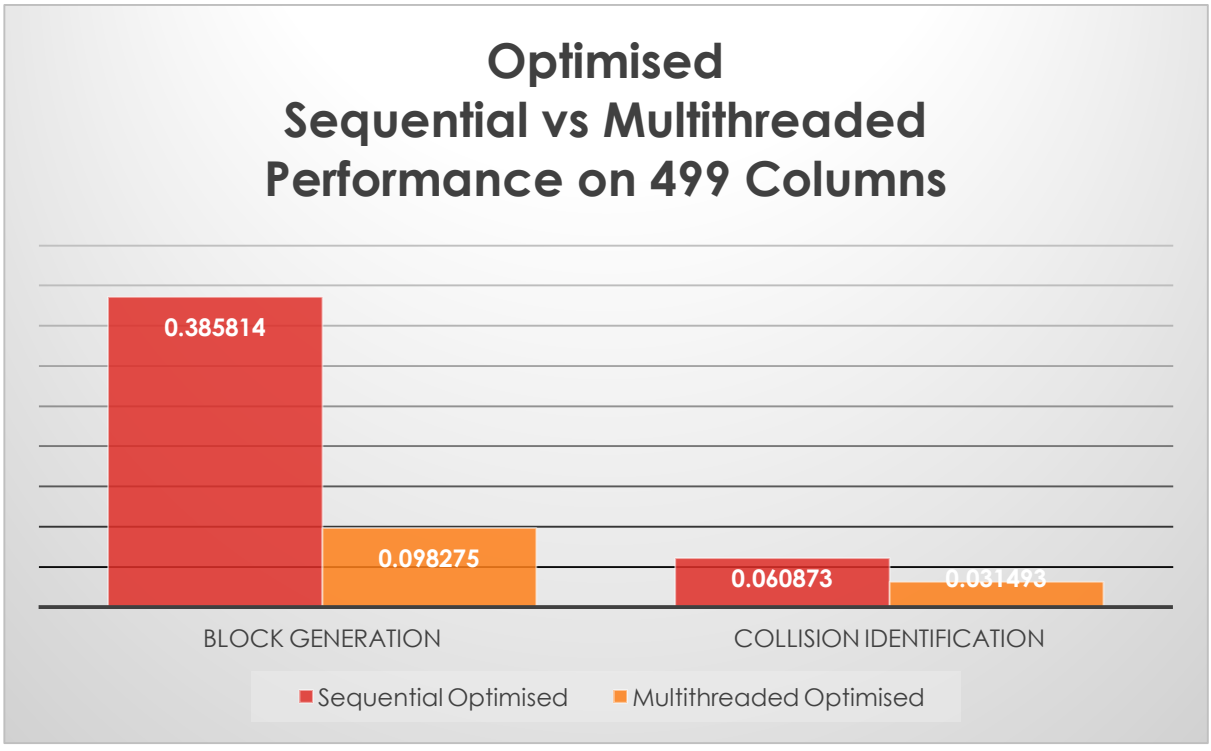
Note that all results documented here are **after** matrix transposition. Also note that **dia** used here is **$2.5 * 10^{-5}$**, rather than the default **dia** of **$1 * 10^{-6}$**

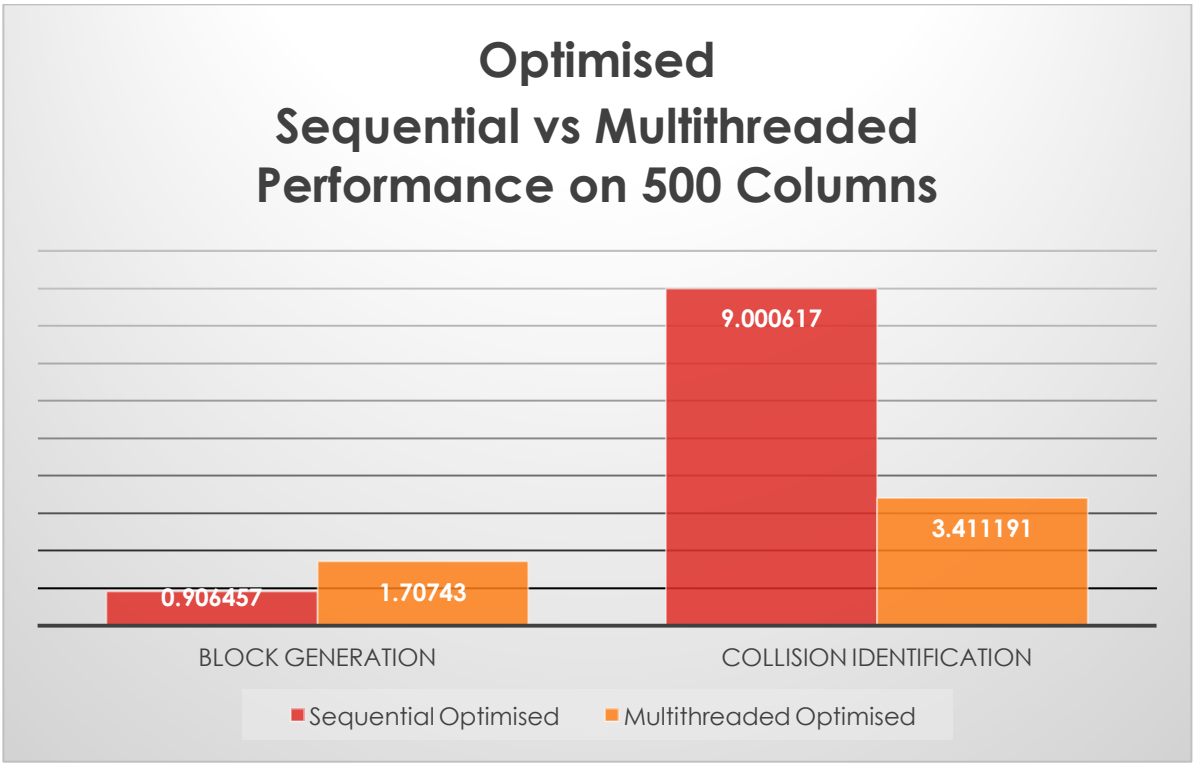
Function	Block Generation (499 columns in seconds)	Collision Identification (499 columns in seconds)	Block Generation (500 columns in seconds)	Collision Identification (500 columns in seconds)
Sequential Brue Force	20.249409	71.753608	163	DNF
Multithreaded Brute Force	4.649090	20.374063	DNF	DNF
Sequential Optimised	0.385814	0.060873	**0.906457**	9.000617
Multithreaded Optimised	**0.098275**	**0.031493**	1.707430	**3.411191**

DNF: Did not finish




Optimised Approaches
EXECUTION TIME IN SECONDS
10
9
8
7
6
5
4
3
2
1
0
Block Generation (499 columns)
Collision Identification (499 columns)
Block Generation (500 columns)
Collision Identification (500 columns)
FUNCTION AND COLLUMNS RUN UPON
Sequential Optimised
Multithreaded Optimised


Optimised
Sequential vs Multithreaded
Performance on 499 Columns
0.385814
0.098275
0.060873
0.031493
BLOCK GENERATION
COLLISION IDENTIFICATION
Sequential Optimised
Multithreaded Optimised



The particular data set provided clearly displayed the real world data problems that multithreaded coding has to deal with; most prominently – load balancing. During the undertaking of this project, the same load-balancing issues were encountered frequently, and were the most challenging issues to approach.

As the only data set available to test the existing code was the provided sample data set, it was impossible to properly stress test the code. An obvious case to test worst case complexity would be if all 500 columns were as densely populated as the 500th column. In this case, any parallelisation by columns would show an appreciable improvement in execution time, in addition to the parallelising within each column.

As a result of the load balancing problems, programmers are forced to (and should always) consider the algorithm with the minimal theoretical worst case complexity. As shown in Table 1, only the functions with optimised algorithms (and lowest big O complexity) were able to finish the large dataset (inclusive of 500th column).

Although multithreading was initially thought to be a trivial task with the use of the OpenMP library, a far greater understanding of omp functions, memory and cache management was required to form correctly behaving parallel code.

In hindsight, time would have been far better reorganised towards forming optimal sequential solutions first, followed by multithreading these approaches. While there is likely a more optimal solution for multi-threading the optimised block generation code, these results serve to show how poor load-balancing has effects on multithreaded programs.

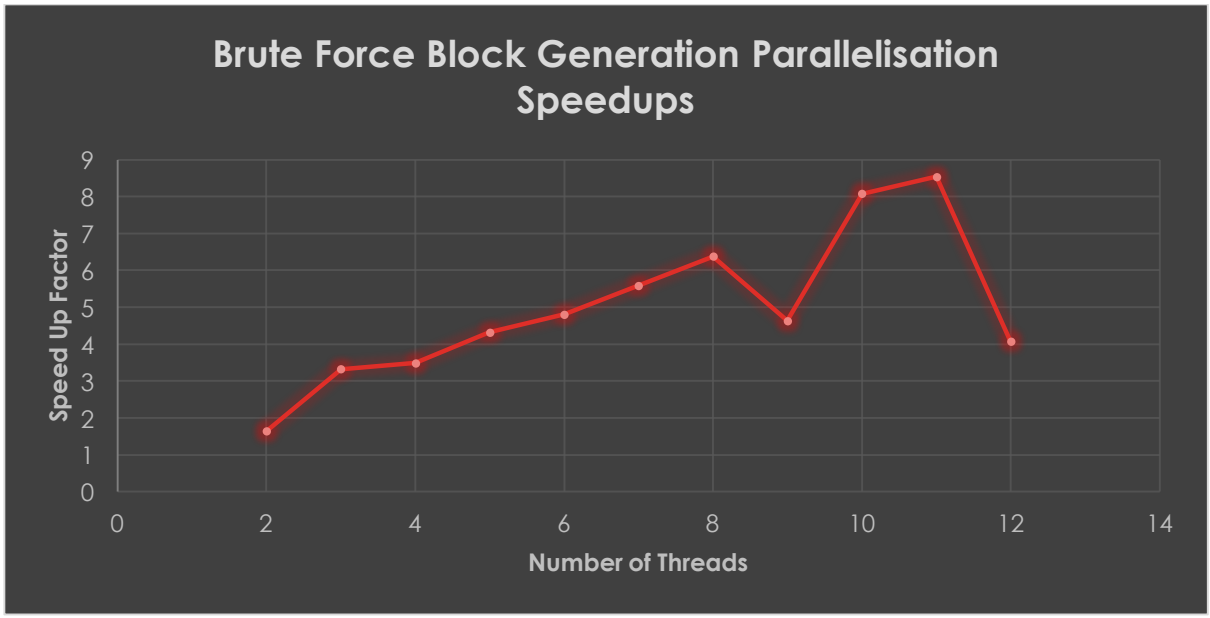
One important set of observations that came out of tackling the project were the substantial differences between performance gains attainable from parallelization, and those attainable from algorithmic improvements. This stems from the fact that an

algorithmic redesign that manages to significantly reduce big O complexity can reduce a program's instruction count drastically. Where a change from O(n^2) to O(nlogn) can speed a program up thousands of times over, depending on the data size, parallelization is limited by the number of available cores, which is typically only 4. Thus one conclusion that can be drawn from our results and experiments is that algorithmic redesign is often a better starting point for improving performance in a program. Parallelization is a powerful tool that can allow a program to make the most of the hardware it's working on. However, it should not overshadow more influential factors of a program's performance, such as the theoretical run-time complexity. In particular, an O(nlogn) algorithm unable to be parallelised will almost always be substantially faster than an embarrassingly parallel $O(n^2)$ algorithm.

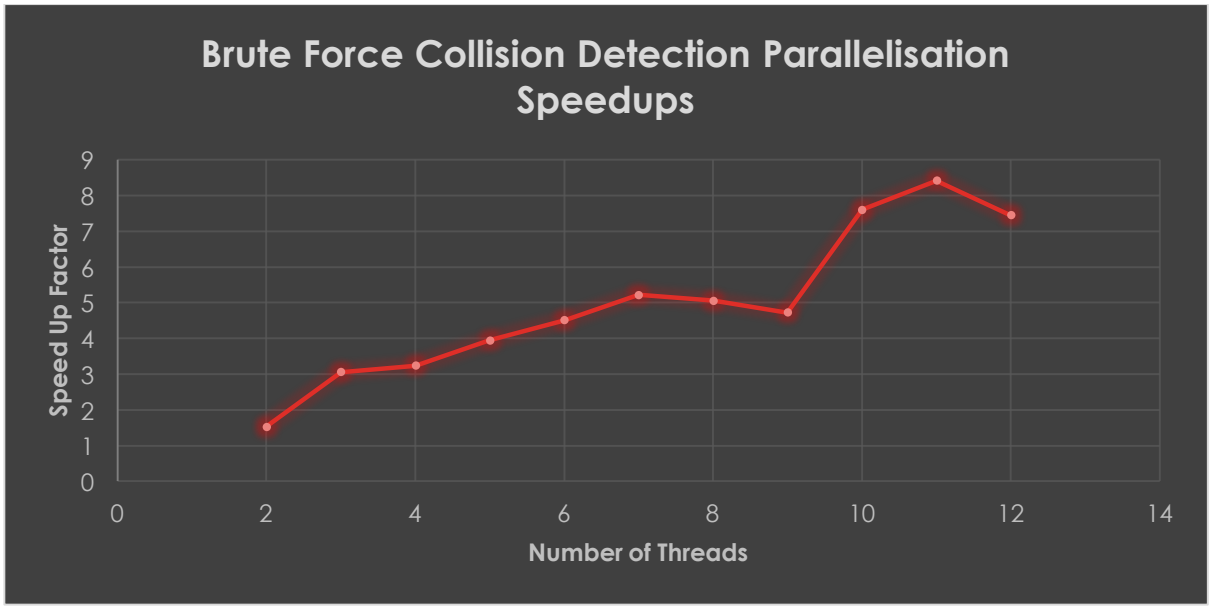
Following on from this point, a general approach to tackling performance issues in a program should start with a thorough analysis of the program's runtime complexity, and then move onto parallelization only after all avenues to improve runtime complexity have been explored. This is likely to result in substantially better performance than an approach that immediately jumps to parallelization, without ever stopping to think about the algorithmic runtime complexity of the code. While brute-force code can often achieve better parallel speed-ups, due to inherently less dependencies (given the simplicity of such code), it is often worth sacrificing some parallelizability if it results in a much better Big-O complexity.

In the future, a simpler approach could be used upon the same problems, with newer functions from OpenMP 4.0 and higher. In addition, a greater understanding of relevant OpenMP functions and constructs could reduce the time taken to multithread any one approach. Having access to some more advanced data structures, such as those available in C++, could assist in designing algorithms of better space complexity, but would be unlikely to improve the overall time complexity (however this has yet to be tested, and may be a path for future investigation, especially in the case of creating custom written hashing functions for unordered sets and maps).

Several experiments were carried out to measure the scalability of the parallel code, and how well the performance gains and speedups scaled with more threads. This was done by performing execution runs on the 12-core cluster machines, with an iteratively increasing number of threads used for the parallel regions. Starting with a thread count of 2, and moving up to 12 threads in increments of 1, the speed-up factor for all versions of the code (brute-force and optimised block generation and collision detection) were measured at each number of threads. Several interesting observations came out this data gathering experiment. The results are depicted below, in graph format:

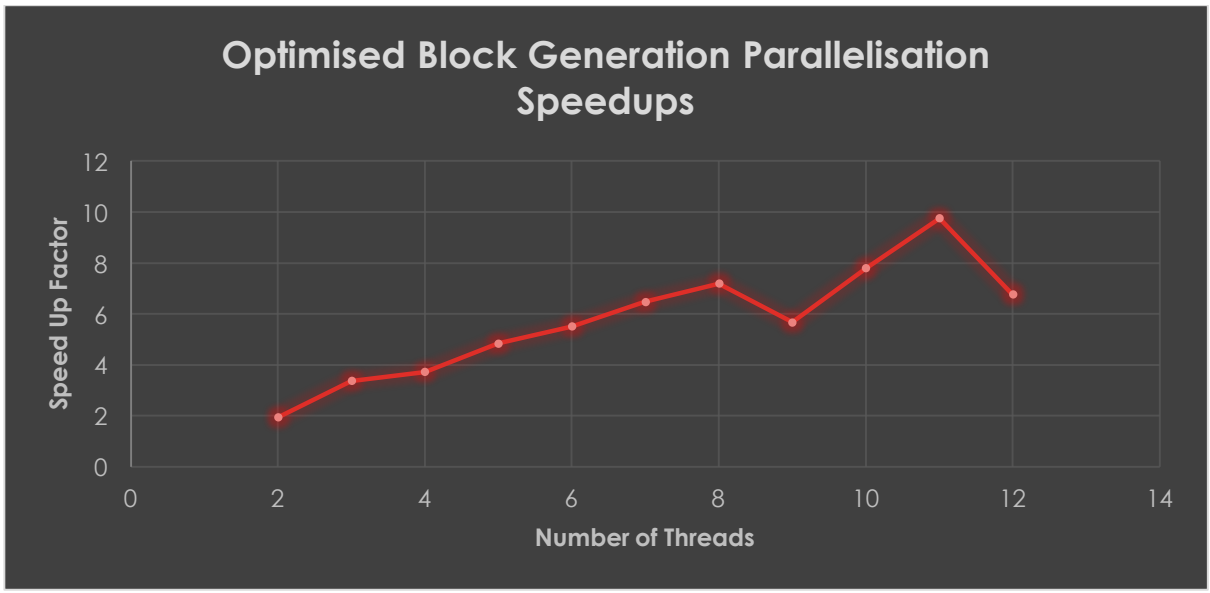


This graph plots the number of threads against the obtained speed-up factor for the brute-force algorithm for block generation. As can be seen, the speed-up factor scales quite well with the number of threads, increasing almost linearly from just under 2 (at 2 threads) to almost 8.5 (at 11 threads). There are some clear dips in the speed-up factor at 9 threads and 12 threads. However, time limitations restricted the number of trial runs that could be executed (only one complete trial of all 12 data sets was performed), thus leaving the possibility that these dips could simply be spurious results. Nevertheless, the clear trend here is a very scalable parallel performance gain.



Similar to the brute-force block generation, the brute-force collision detection's performance also scales remarkably well with the number of active threads, increasing fairly consistently from a speed-up factor of just under 2 (at 2 threads) to a speed-up factor of almost 8.5 (at 11 threads). Once again, notable dips in performance can be observed at 9 threads and 11 threads (as the tests were all run together), however a lack

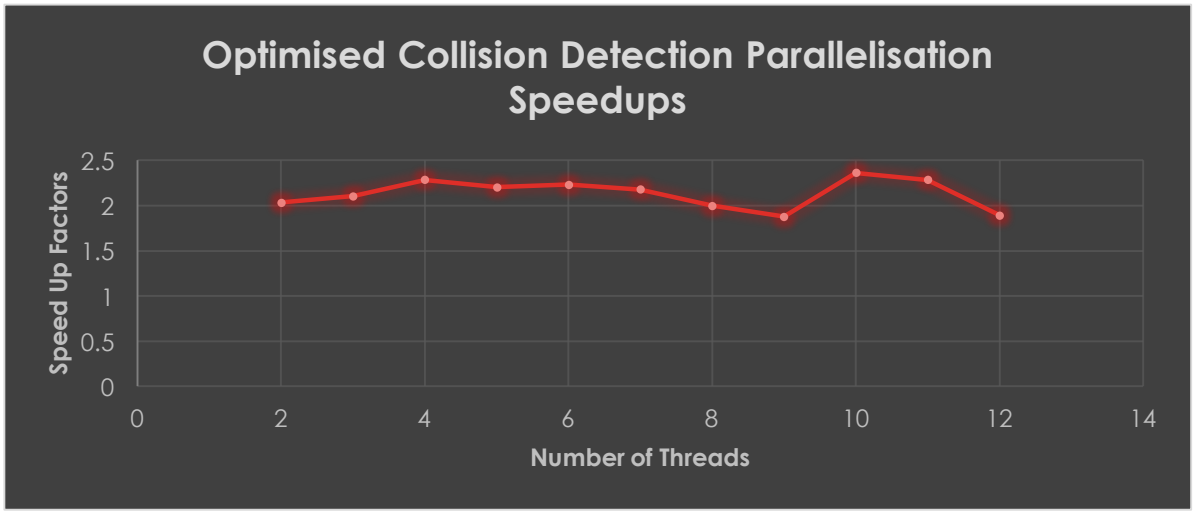
of resources and a lack of time limited our ability to examine the underlying reasons behind these dips.



Putting aside the aforementioned dips (which may perhaps be outliers), the performance of the optimised sequential code scaled remarkably well. The speed-up factors obtained for this algorithm scaled almost linearly from 2 through to 10.

The underlying reasoning behind the remarkable scalability displayed above is the inherently parallelizable nature of the algorithms. All the above algorithms have a key factor in common: their outermost loops have iterations that are independent of one another. This allowed for extremely effective parallelization through the use of simple work-sharing constructs, by splitting the outermost loop roughly evenly among all active threads. It is thus natural (given the almost embarrassingly parallel nature of the algorithms), that the performance would scale this well.

In contrast, the parallelization of the optimised collision detection algorithm was a much more intricate task. There were several complexities (described in section 4.4) that impeded regular attempts at parallelization. A more intricate solution involving tasks was required to attain an effective speed-up. Naturally, the collision detection displayed significantly less scalability as a result.



As can be seen, the speed-up factor for the optimised collision detection remained fairly constant at about 2-2.5, and had no observable relationship with the number of active threads. One likely reason for this is that the collision detection, with the new optimised algorithm, is a very small process. The bottleneck (in terms of theoretical runtime complexity) in the collision detection process is actually the O(nlogn) sorting step that is performed on the array of blocks. This step is run completely sequentially. The only parallelized portion of the optimised collision detection is the O(n) loop through the array of blocks; thus the runtime is going to be dominated by the sequential sorting step anyway. Additionally, the O(n) loop through the blocks, with our data set (DIA of $2.5*10^{-6}$) only runs over an array of roughly 240,000 blocks. This is an incredibly small dataset for a modern computer to process in linear time. As such, the primary reason behind the constant speed-up factor is that the parallelized O(n) computation stage actually finishes fast enough that the overheads involved in parallelization damage the scalability of the parallelized code. For example, considering how the parallel algorithm works (using tasks, as described in section 4.4), the more threads there are, the smaller each task will be. Considering a situation when 12 threads are running, each task will involve processing roughly 20,000 blocks (240,000/12) in linear time. Such computation is almost trivial on modern hardware. As a result, the more threads there are, the more trivial the computation in individual tasks become, and the more damaging the critical region (used to merge partial thread-private collision databases with the collective shared collision database) at the end of each task becomes to performance. Given these factors, it is thus well within expectations that the collision detection scales poorly with more threads.

As a final remark, one factor that has not been considered here is the time taken for I/O, including reading in all relevant data, and transposition of the matrix. The time taken for I/O averages at about **0.65** seconds for this program. Compared to the times taken for the optimised algorithms shown above (especially the parallel versions), this is extremely significant, and could constitute the majority of the program's run-time. The I/O cannot be parallelized, and is thus likely to be the bottleneck in performance in a real execution of this program. This strongly reflects the notion that the performance of a parallel program is severely limited by the performance of its sequential part, and the fraction of the program that is sequential. If the entire program was timed (using only the optimised algorithms and their parallel counterparts), the speed-up would be limited by the I/O bottleneck. This bottleneck would become less and less significant as DIA was increased, and there was more computation work to be performed in parallel. This idea is a concrete example of Amdahl's Law, which states that the performance of a parallel program is limited by the performance of its sequential components, and the overall fraction of the entire program such sequential components represent.